Improving the Performance of Shared-memory Parallel Java Codes Using Application/Kernel Cooperation

Jordi Guitart, Xavier Martorell, Jordi Torres and Eduard Ayguadé

European Center for Parallelism of Barcelona (CEPBA)
Computer Architecture Department
Technical University of Catalonia
C/ Jordi Girona 1-3, Campus Nord UPC, Mòdul C6
E-08034 Barcelona (Spain)

{jguitart, xavim, torres, eduard}@cepba.upc.es

ABSTRACT

In this paper we propose mechanisms to improve the performance of parallel Java applications executing on multiprogrammed shared-memory multiprocessors. The proposal is based on a dialog between each Java application and the underlying execution environment (mainly the resource manager in the kernel) so that both cooperate on improving the overall performance (individual application speedup and system throughput). This dialog implies modifications at the application (or compiler), the threading library and kernel levels. Although performance degradation of parallel applications running on multiprogrammed systems has been analyzed and addressed using some kind of cooperation on other environments, in this paper we intend to detect this problem in the Java context, determine if cooperation is also a good mechanism to improve performance, and in this case, which modifications are required to the Java execution environment to allow this cooperation. The paper includes experimental results based on parallel OpenMP-like workloads, including both applications able to cooperate (malleable) and not (non-malleable).

1. INTRODUCTION

Over the last few years, Java has emerged as an interesting language for the network programming community. This has largely occurred as a direct consequence of the design of the Java language. This design includes, among others, important aspects such as portability and architecture neutrality of Java code, and its multithreading facilities. The latter is achieved through built-in support for threads in the language definition. The Java library provides the Thread class definition, and Java runtime provides support for thread and monitor primitives. These characteristics, besides others like its familiarity (due to its resemblance with C/C++), its robustness, security and distributed nature also make it a potentially interesting language for parallel scientific computing.

The use of Java for high-performance scientific applications faces a number of problems that are currently subject of research. The first one is related with the large overheads incurred by the Java implementation, which favor equivalent versions of the same application written in other languages such as Fortran or C. The second problem is related with the lack of programming models for the specification of parallelism. This makes difficult the parallelization of Java applications because it implies the explicit management of the parallelism/synchronization. And third, the threading support available in the JVM incurs large overheads that can easily offset any gain due to parallel execution. Recently, results show how the performance gap between Java and other traditional languages is being reduced [2], and some language extensions [3] and runtime support have been proposed [12] to ease the specification of Java parallel applications and make execution more efficient.

In this paper we analyze causes of performance degradation when parallel Java applications are executed in a multiprogrammed environment. The main issues that lead to this degradation can be summarized as follows:

- The Java runtime environment does not allow applications to have control on the number of kernel threads where Java threads map and to make suggestions about the scheduling of these kernel threads.
- The Java runtime environment does not inform the application about the dynamic status of the underlying system so that the application cannot adapt its execution to these characteristics.
- Large number of migrations of the processes allocated to an application, due to scheduling polices that do not consider parallel Java applications as an allocation unit.

Table 1. LUAppl performance degradation

Java Threads	Active Kernel Threads	Execution Time in seconds
3	**3**	39,7
4	**4**	34,3
4	**3**	44,1

In order to motivate our analysis and proposal, we have designed a simple experiment based on LUAppl, a LU reduction kernel over a two-dimensional matrix of double-precision elements taken from [12] that uses a matrix of 1000x1000 elements. The experiment consists of a set of executions of LUAppl running with different number of Java threads and active kernel threads (with a processor assigned to them). Table 1 shows the average execution time on a SGI Origin 2000 architecture [17] with MIPS R10000 processors at 250 MHz running JVM version Sun Java 1.2.2. The first and second rows show that when the number of Java threads matches the number of active kernel threads, the application benefits from running with more threads. However, if the number of active kernel threads provided to support the execution does not match, as shown in the third row, the performance is degraded. In this case the execution environment (mainly the resource manager in the kernel) is providing only three active kernel threads, probably because either there are no more processors available to satisfy the application requirements, or the execution environment is unable to determine the concurrency level of the application. In the first case, this situation results in an execution time worse than the one achieved if the application would have known that only three processors were available and would have adapted its behavior to simply generate three Java threads (as shown in the first row). In the second case, this situation results in an execution time worse than the one achieved if the execution environment would have known the concurrency level of the application and would have provided four active kernel threads (as shown in the second row).

Although performance degradation of parallel applications running on multiprogrammed systems has been analyzed and addressed using some kind of cooperation on other environments, in this paper we intend to detect this problem in the Java context, determine if cooperation is also a good mechanism to improve performance, and in this case, which modifications are required to the Java execution environment to allow this cooperation.

We have found two different ways of approaching the problem in the Java context. In the first one, we simply use one of the services supplied by the Java native underlying threads library to inform the library about the concurrency level of the application. In the second one, we propose to execute Java applications on top of JNE (Java Nanos Environment built around the Nano-threads environment [11]). JNE provides the mechanisms to establish a bi-directional communication path between the application and the underlying system.

The rest of the paper is organized as follows: Section 2 presents the related work. Section 3 analyzes the possibility of informing the system about the concurrency level of the application. Section 4 presents an overview of the JNE and Section 5 reveals some details of its implementation. The approach presented in these two sections is based on malleable applications and a dialog between the applications and the execution environment. Section 6 presents the evaluation of this proposal, and finally, Section 7 presents the conclusions of this paper.

2. RELATED WORK

Experience on real systems shows that with contemporary kernel schedulers, parallel applications suffer from performance degradation when executed in an open multiprogrammed environment. As a consequence, intervention from the system administrator is usually required, in order to guarantee a minimum quality of service with respect to the resources allocated to each parallel application (CPU time, memory etc.). Although the use of sophisticated queuing systems and system administration policies (HP-UX Workload Manager [18], IBM AIX WLM [7], Solaris RM [20], IRIX Miser Batch Processing System [16], …) may improve the execution conditions for parallel applications, the use of hard limits for the execution of parallel jobs with queuing systems may jeopardize global system performance in terms of utilization and fairness.

Even with convenient queuing systems and system administrator's policies, application and system performance may still suffer because users are only able to provide very coarse descriptions of the resource requirements of their jobs (number of processors, CPU time, etc.). Fine–grain events that happen at execution time (spawning parallelism, sequential code, synchronizations, etc.), which are very important for performance, can only be handled at the level of the runtime system, through an efficient communication interface with the operating system.

The NANOS RM [10] is an application-oriented resource manager, i.e. the unit of resource allocation and management is the parallel application. Other resource managers (like the Solaris RM or the AIX WLM) work at workload or user granularity. Having parallel applications as units for resource management allows the application of performance-driven policies [4][5] that take into account the characteristics of these applications (e.g. speed-up or efficiency in the use of resources).

The NRM takes decisions at the same level than the kernel does. This means that it does not only allocates processors to a particular application, but also it performs the mapping between kernel threads and processors and controls the initial memory placement. This is an issue that is important to consider in the Java environment using the native threads model (several kernel threads in contraposition to the green threads model that just uses one kernel thread for all the Java threads in the application). Our implementation targets the SGI IRIX JVM, which like many others (Linux, Solaris, Alpha, IBM, …), implements the native threads model using the Pthreads [15] library. Thus, one Java thread maps directly into one pthread, and the Pthreads library is responsible for scheduling these pthreads over the kernel threads offered by the operating system. Like we have said before, this situation can lead to performance degradation of parallel Java applications due to the lack of control about the number of kernel threads created and the lack of information about available processors.

The Jikes RVM [1] implements a different thread model. It provides virtual processors in the Java run-time system to execute the Java threads. Usually, there are more Java threads than virtual processors. Each virtual processor is scheduled onto a pthread. This means that, as the other threads models do, Jikes relies on the Pthreads library for scheduling the pthreads over the kernel threads offered by the operating system, suffering of the same problems about performance degradation of parallel Java applications.

3. CONCURRENCY LEVEL

Current implementation of SGI IRIX JVM version Sun Java 1.2.2 does not inform the underlying threads layer about the desired concurrency level of the application. By default, the threads library adjusts the level of concurrency itself as the application runs using metrics that include the number of user context switches and CPU bandwidth. In order to provide the library with a more accurate hint about the concurrency level of the application, the programmer could invoke, at appropriate points in the application, the `pthread_setconcurrency(level)` service of the Pthreads library. The argument `level` is used by Pthreads to compute the ideal number of kernel threads required to schedule the available Java threads.

Previous experimentation has revealed that informing to the threads library about the concurrency level of the application may have an important incidence on performance. The improvements that we have experimented range from 23% to 58% when executing applications that create threads with a short lifetime. Threads are so short that the threads library is unable to estimate the concurrency level of the application and provide it with the appropriate number of kernel threads. When a hint of the concurrency level is provided by the application, the underlying threads library is capable of immediately providing the necessary kernel threads. A detailed explanation of the effects of the concurrency level on performance for this kind of applications can be found in [8].

For those parallel Java applications that create threads with a long lifetime, like the ones used in this paper, informing about the concurrency level has less impact on performance. For this kind of applications, the threads library has time enough to estimate and adjust the number of kernel threads required during the thread lifetime. However, the time required by the threads library to estimate the concurrency level of the application is not negligible and may approach the order of hundreds of milliseconds (even a few seconds depending of the application). Therefore, providing this hint is beneficial in any case.

In summary, this approach only solves one of the problems mentioned in Section 1. Applications have more control on how Java threads map onto kernel threads because they specify the number of processors required at any time. However, other problems remain open. For instance, this approach does not allow applications to decide about the scheduling of kernel threads. Although it allows applications to inform to the execution environment about their processor requirements, the execution environment cannot inform each application about the number of processors actually assigned to it. In other words, the communication path only exists in one direction. As a consequence, applications cannot react and adapt their behavior to the decisions taken by the underlying system. If informed, applications would be able to restrict themselves in terms of parallelism generation, thus avoiding unnecessary overheads.

Newer versions of the SGI IRIX JVM (from Sun Java 1.3) incorporate this approach, but setting the concurrency level to the maximum number of processors available in the system. So they obtain performance gains similar to the ones obtained with the concurrency level approach, but the problems commented in the previous paragraph remain unresolved.

4. JAVA NANOS ENVIRONMENT

JNE (Java Nanos Environment) is a research platform that provides additional mechanisms to improve the communication between the Java parallel applications and the underlying execution environment, thus allowing applications to collaborate in the thread management.

4.1 Adaptive Java Applications

The first issue considered in JNE is the capability of Java applications to adapt their behavior to the amount of resources offered by the execution environment (malleability [6]). The process is dynamic and implies three important aspects:

- First, the application should be able to request and release processors at any time. This requires from the execution environment an interface to set the number of processors the application wants to run.
- Second, the amount of parallelism that the application will generate (at a given moment) is limited by both the number of processors assigned to the application and the amount of work pending to be executed. The execution environment has to provide an interface to allow the application to check the number of processors available just before spawning parallelism.
- And third, the application should be able to react to processor preemptions and allocations resulting from the operating system allocation decisions. This also requires mechanisms that allow the application, once informed, to recover from possible processor preemptions.

4.2 Application/JNE Interface

Each Java application executing on the JNE shares information with the execution environment. The information includes the number of processors on which the application wants to run at any moment and the number of processors currently allocated by the execution environment to the application.

We have defined a Java class called `jne`, which contains the following two Java methods for calling, through the Java Native Interface (JNI) [19], the JNE services for requesting and consulting processors:

- `cpus_current`: method for consulting the current number of processors allocated to the invoking application.
- `cpus_request(num)`: method for requesting to the execution environment `num` processors.

4.3 JNE Scheduler

The JNE scheduler is based on the NRM mentioned in Section 2. It is responsible for the distribution of processors to applications. At any time, there is a current active scheduling policy that is applied to all applications running in the system. The scheduler observes application demands, estimates the load of the machine, and finally distributes processors accordingly. The scheduler also decides which processors are assigned to each application taking into account data affinity issues (i.e. helping the application to exploit data locality whenever possible).

JNE offers a set of scheduling policies, including batch, round robin, equipartition and others than combine space- and time-sharing. In our evaluation we use Dynamic Space Sharing (DSS) [13][14]. In DSS, each application receives a number of processors that is proportional to its request and inversely proportional to the total workload of the system, expressed as the sum of processor requests of all jobs in the system.

4.4 JOMP Applications

As part of our research platform, we are using the JOMP [3] compiler developed at EPCC. JOMP includes OpenMP-like extensions to specify parallelism in Java applications using a shared-memory programming paradigm. The programming model is based on a set of directives and library methods. The compiler automates the generation of parallelism from the directives specified by the user. For our research, we have modified the implementation of the JOMP compiler and supporting runtime library to implement the communication between the application and JNE.

JOMP applications will allow us to generate the communication mechanisms transparently to the user and to confirm for this kind of applications the performance improvements obtained in Java applications parallelized using directly the Java thread support [8].

Figure 1a shows the source code of a simple JOMP program with a `parallel` directive. This means that all the threads in the team will concurrently execute the code encapsulated by this directive. The directive has also a `private` clause for the `myid` variable, informing the compiler to allocate a private copy of this variable for each thread (usually in the stack of the encapsulating method).

Figure 1b shows the code transformation made by the JOMP compiler highlighting the utilization of one of the JNE interface services. Notice that on

```
public class Hello {
  public static void main (String argv[]) {
    int myid;

    //omp parallel private (myid)
    {
      myid = OMP.getThreadNum();
      System.out.println("Hello from" + myid);
    }
  }
}
```

(a) original code

```
public class Hello {
  public static void main (String argv[]) {
    int myid;

    // OMP PARALLEL BLOCK BEGINS
    jomp.runtime.OMP.setNumThreads(jne.cpus_current());
    __omp_Class0 __omp_Object0 = new __omp_Class0();
    __omp_Object0.argv = argv;
    try {
      jomp.runtime.OMP.doParallel(__omp_Object0);
    } catch(Throwable __omp_exception) {
        System.err.println("OMP Warning:
          Illegal thread exception ignored!");
        System.err.println(__omp_exception);
    }
    argv = __omp_Object0.argv;
    // OMP PARALLEL BLOCK ENDS
  }
 // OMP PARALLEL REGION INNER CLASS DEFINITION BEGINS
 private static class __omp_Class0 extends jomp.runtime.BusyTask {
   String [ ] argv;

   public void go(int __omp_me) throws Throwable {
     int myid;

     // OMP USER CODE BEGINS
     myid = OMP.getThreadNum();
     System.out.println("Hello from" + myid);
     // OMP USER CODE ENDS
   }
 }
 // OMP PARALLEL REGION INNER CLASS DEFINITION ENDS
}
```

(b) transformed code

Figure 1. Simple example showing the use of the JNE interface for JOMP applications

encountering a parallel directive, the compiler creates a new class that extends the `BusyTask` class. The new class has a `go()` method, containing the code inside the parallel region, and declarations of private variables like `myid`. The new class contains also data members corresponding to reduction and shared variables like `argv`.

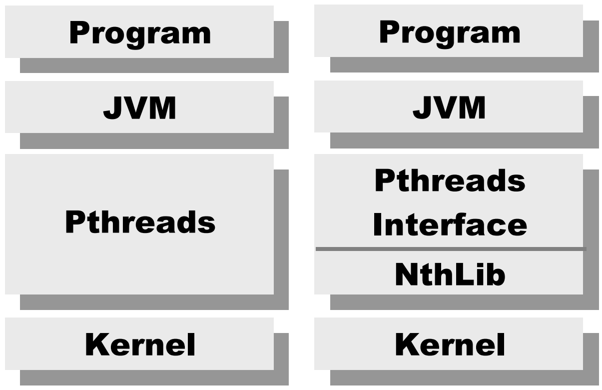
The JOMP runtime library has been modified so that, when an application starts, it requests as many processors for this application as indicated in one of the arguments of the interpreter command line (`-Djomp.threads`). This request is made using the `cpus_request()` method available in the JNE interface.

After that, every time the application has to spawn parallelism (i.e. when encountering a `parallel` directive) the compiler injects a call to `cpus_current()` method from the JNE interface to check the number of processors currently allocated to the application, as shown in Figure 1b. With this information, the application generates work for as many threads as processors available to run by invoking the `setNumThreads()` method and causes the execution of the `go()` method on each thread in the team, calling the `doParallel()` method from the JOMP runtime library.

The user can change the concurrency level of the application at any time (whenever there is a change in the parallelism profile) invoking the `setNumThreads()` method from the JOMP runtime library. In this case, in order to inform the execution environment about the new processor requirements of the application, the JOMP compiler would replace this invocation with one to the `cpus_request()` method from JNE interface.

5. IMPLEMENTATION OF JNE

Like it has been said in Section 2, the JVM implementation of SGI IRIX implements the native threads model using the Pthreads [15] library (Figure 2a).



(a) (b)

Figure 2. (a) Java IRIX Environment (b) Java Nanos Environment

In order to implement the mechanisms described in Section 4, we decided to replace the Pthreads library with the Nano-threads Library (NthLib) [11]. NthLib[1] is a user level threads package specially designed for supporting parallel applications and providing a communication path between application and execution environment. In order to avoid modifications of the JVM, we maintain the Pthreads library interface but we have rewritten the library methods using the services provided by NthLib (Figure 2b).

NthLib provides the following services:

- **Thread management services:** nth_create (create a nano-thread), nth_exit (finalize a nano-thread), nth_wait (block a nano-thread) and nth_yield (yield virtual processor to another nano-thread).
- **Generic queue management services:** nth_queue_init (initialize queue), nth_enqueue (enqueue nano-thread on queue) and nth_dequeue (dequeue nano-thread from queue).

---

[1] Although NthLib uses IRIX *sprocs* instead of the IRIX *nsprocs* used by Pthreads, we have made experiments showing that this fact has not noticeable effect on performance.

- **Ready queue management services:** nth_to_rq_end (enqueue nano-thread on global ready queue) and nth_to_lrq_end (enqueue nano-thread on local ready queue).
- **Mutual exclusion services:** spin_init (initialize spin lock), spin_lock (lock spin) and spin_unlock (unlock spin).

```
int pthread_cond_signal(pthread_cond_t
*c) {
 struct nth_desc *nth_id;
 /* dequeue first thread waiting in
    condition variable */
 nth_dequeue(cond_queue(c),nth_id);
 /* add dequeued thread to ready
    queue */
 if(nth_id != NULL)
     nth_to_rq_end(nth_id);
 return 0;
}
```

Figure 3. Replacement example

Figure 3 shows a simplified example showing the replacement of the `pthread_cond_signal` method, emphasizing the use of NthLib services.

The JNE scheduler is implemented as a user-level process that wakes up at a fixed time quantum, examines the current requests of the applications and distributes processors, applying a scheduling policy. With this configuration, direct modification of the native kernel is not required to show the usefulness of the proposed environment.

6. EVALUATION

In this section, we present the experimental platform that we have used to conduct our evaluation and the main conclusions drawn from our initial experimentation with JNE. The current implementation targets the SGI Origin 2000 architecture [17] with MIPS R10000 processors at 250 MHz running the JVM version Sun Java 1.2.2.

We have used the Java Grande Benchmarks [9] to evaluate JNE. These benchmarks can be found in three different versions (sequential, multithreaded and JOMP), with three different sizes (A, B and C). We have used the JOMP version (size B).

Although JNE has been developed to improve performance of malleable applications (that is, applications able to adapt their behavior to the amount of resources offered by the execution environment), we want to support the efficient execution of non-malleable applications too, which are common (and often it is not easy convert them to malleable). For example, in the JOMP version of the Java Grande Benchmarks, only SOR, LUFact and Euler are malleable. Crypt, Series, MonteCarlo and RayTracer are not malleable because they only have one parallel region and, as we have commented before, adaptability is achieved at the beginning of each parallel region. SparseMatmult is not malleable because the concurrency level of the application is used as size in some data structures, making impossible to change dynamically this value without modifying the application.

We have decided to make experiments with malleable applications based only on SOR and LUFact because Euler is unstable when running on IRIX (it provides incorrect results in some executions). SOR performs 100 iterations of a successive over-relaxation on an N x N grid (N = 1500). LUFact solves an N x N linear system using LU factorization followed by a triangular solver (N = 1000).

On the other side, our experiments with non-malleable applications are based on Crypt and Series. Crypt performs IDEA (International Data Encryption Algorithm) encryption and decryption on an array of N bytes (N = 20000000). Series computes the first N fourier coefficients of the function f(x) = (x+1)^x on the interval 0,2 (N = 10000).

All the experiments in this paper have been performed in the so-called *cpusets* in IRIX. A *cpuset* consists of a set of dedicated processors in a multiprogrammed machine. However, although a number of processors is reserved for running some applications, other resources (like the interconnection network or the memory) are shared with the rest of applications running in the system. This sharing can interfere the behavior of the applications running inside the *cpuset* and produce noticeable performance degradation. Although it is difficult to quantify (and predict) it, we have observed degradation around 10% depending on the system load and the application characteristics (a memory intensive application will be more interfered than an application with low memory use). This effect can be attenuated in individual executions incrementing the number of measurements and discarding anomalous values. But when executing the applications as a part of a workload any interference could affect their performance in the percentage commented.

6.1 Single Application Performance

In our first set of experiments, an individual instance of SOR, LUFact, Crypt and Series is executed inside a *cpuset*, in its sequential version and its JOMP version with different concurrency levels (between 1 and 16 threads). With this experiment we intend to evaluate the impact on performance of the Pthreads library replacement by the NthLib, and analyze the scalability of each application.

The speedup obtained for SOR, LUFact, Crypt and Series running with different concurrency levels with respect to the sequential version is plotted in Figures 4, 5, 6 and 7, respectively. For each experiment, 10 executions have been performed. The first series (labeled IRIX) corresponds to the execution on the native IRIX system. The second series (labeled IRIX+SETC) corresponds to the execution on the native IRIX system when the application informs to the Pthreads library about its concurrency level (using the `pthread_setconcurrency` mechanism described in Section 3). The third series (labeled JNE–CPUM) corresponds to the execution time on top of the JNE with the JNE scheduler not active. And the fourth series (labeled JNE+CPUM) corresponds to the execution time on top of the JNE with the JNE scheduler active.

From the analysis of the speedup figures of malleable applications (SOR and LUFact, Figure 4 and Figure 5, respectively) we can derive four important conclusions. First, the performance obtained running with IRIX is very low, due to the large number of process migrations occurred. For example, for LUFact with concurrency level of 8 threads the system performs 9,6 process migrations per second on average. An important

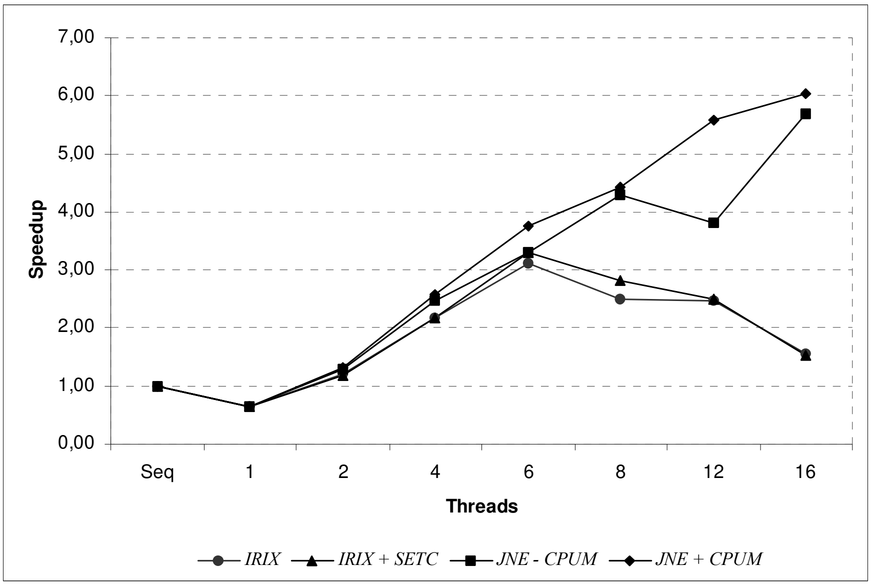


Figure 4. SOR standalone speedup

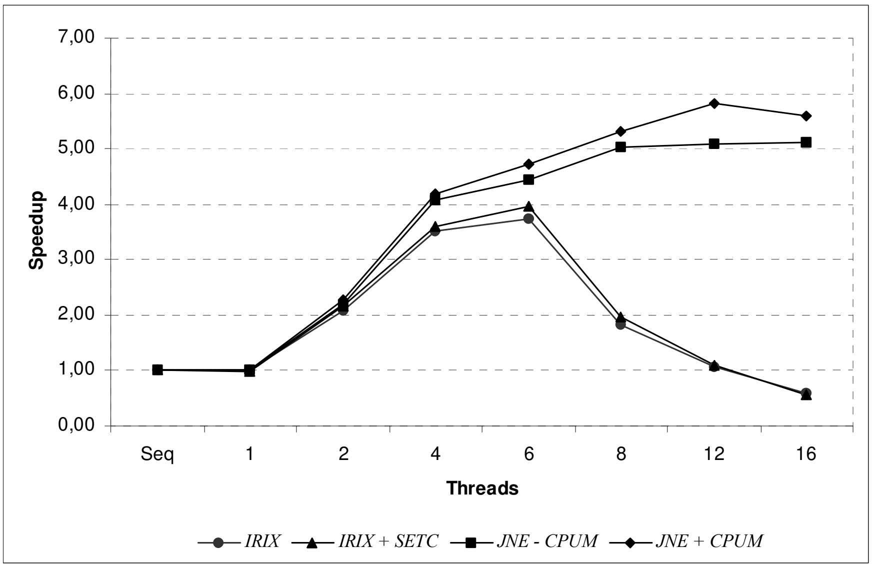


Figure 5. LUFact standalone speedup

part of these process migrations are produced when application invokes the `yield()` method. The Pthreads library does not try to exploit any data affinity in this point, and relies on the underlying operating system to perform the yield operation. This increases the process migrations and reduces data affinity. This problem acquires special relevance in JOMP applications (especially when they have several parallel regions), which frequently use the `yield()` method (when threads look for new work to be executed or when threads wait for a barrier to be opened), like many others runtimes do, to implement a polite scheduling that allows others threads to execute when there is not useful work to do.

The second conclusion is that, like we have advanced in Section 3, improvements on performance when running with IRIX+SETC are not very high, because the JOMP runtime creates threads at user level with a long lifetime. However, the large number of migrations performed by IRIX is still the main cause of the bad behavior.

The third conclusion is that running with JNE–CPUM provides noticeable performance improvements that can be explained as follows. NthLib tries to exploit data affinity itself at nano-thread level. When a thread invokes the `nth_yield()` method, it yields its kernel thread to another nano-thread and enqueues itself in the local ready queue of this kernel thread. In this way, data affinity at nano-thread level is improved and the yield operation is accomplished avoiding unnecessary operating system intervention, reducing the number of process migrations. For example, for LUFact with concurrency level of 8 threads the system performs 1,4 process migrations per second on average.

Notice that JNE–CPUM does not bind kernel threads to processors in the *cpuset*. This explains the anomalous behavior observed for 6 and 12 threads. In both cases, the application is executed in a *cpuset* larger than the number of processors required (*cpuset* of 8 processors and *cpuset* of 16 processors, respectively). This means that there are free processors, and as kernel threads are not bound with processors, migrations are incremented (11,6 migrations per second on average when executing SOR with concurrency level of 12 threads).

Finally, the fourth conclusion of this set of experiments is that running with JNE+CPUM

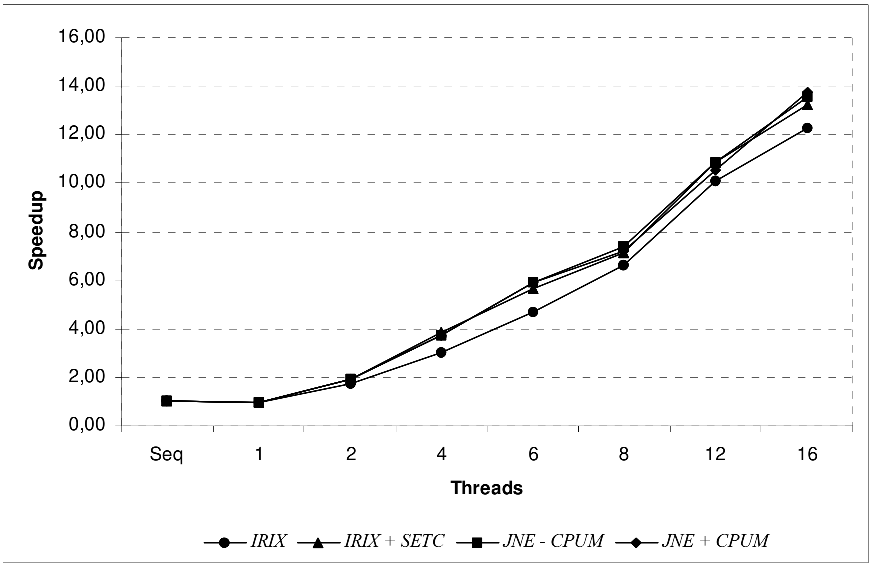


Figure 6. Crypt standalone speedup

improves the performance even more. In addition to all the advantages of the JNE–CPUM approach, the JNE scheduler strengthens data affinity at kernel thread level by binding kernel threads to the processors assigned to the application. This binding totally eliminates process migrations.

The low scalability achieved in these applications can be explained because SOR and LUFact have 1 parallel region repeated several times inside a time step loop. This means that work generation and thread synchronization are done several times, both facts producing considerable overhead. Besides, threads reuse data at every parallel region, so process migrations can heavily affect performance because data affinity is lost.

On the other side, the analysis of speedup figures of non-malleable applications (Crypt and Series,

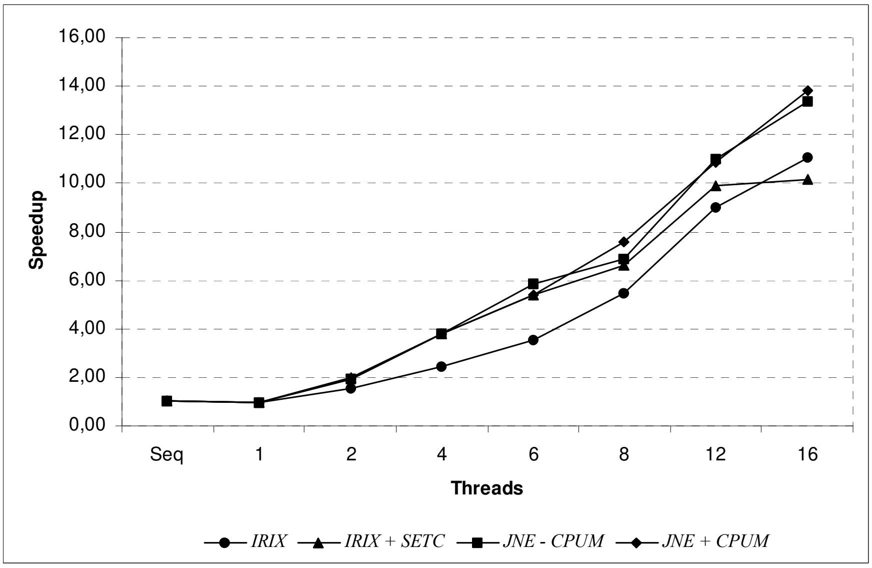


Figure 7. Series standalone speedup

Figure 6 and Figure 7, respectively) reveals us that all the approaches evaluated obtain similar performance, achieving good scalability (nearly linear). Only when running with IRIX the speedups obtained are a little bit worse because the execution environment (Pthreads library in this case) needs some time to estimate the concurrency level of the application, how it has been explained in Section 3. Notice that, if we inform the execution environment about this concurrency level, as it is done in the other approaches, performance is improved.

The high scalability achieved in these applications can be explained because Crypt and Series have only one parallel region. This means that work generation and thread synchronization are done only once, minimizing the overhead produced by these actions. Besides, threads do not reuse data, so process migrations are not critical for performance.

6.2 Multiprogrammed Workloads

6.2.1 Malleable applications

For our second set of experiments, we have defined a workload composed of an instance of LUFact with concurrency level of 2 threads, an instance of SOR with concurrency level of 4 threads, an instance of LUFact with concurrency level of 4 threads and an instance of SOR with concurrency level of 6 threads. These applications instances are simultaneously started inside a *cpuset* with 16 processors, and they are continuously restarted until one of them is repeated 10 times. Notice that the system is not overloaded (i.e. the number of processors in the *cpuset* is greater or equal than the maximum load). With this experiment we intend to evaluate the performance of JOMP malleable applications when running in a non-overloaded multiprogrammed environment.

Figure 8 draws the speedup obtained for each application instance in the workload relative to the sequential version. The first and second series have the same meaning as in the first set of experiments. The third series (labeled JNE not mall) corresponds to the execution time on top of the JNE with the JNE scheduler active using a DSS scheduling policy, assuming that applications do not use the JNE interface to adapt themselves to the available resources (they behave as non-malleable applications). And the fourth series (labeled JNE mall) corresponds to the execution time on top of the JNE when applications use the JNE interface to adapt themselves to the available resources.

Since the system is not overloaded, each

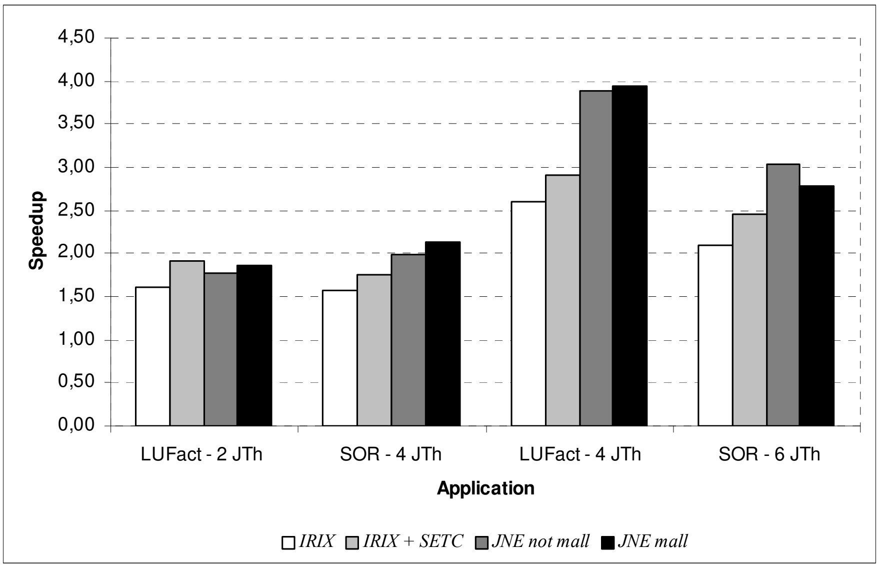


Figure 8. Speedup of each application instance in the first workload

application instance should be able to run with as many processors as requested. Therefore, the speedup should be the same as if executed alone in the *cpuset*. However, speedup figures are worse than one could expect.

First, when executing with IRIX, the speedup achieved is very low. This is caused by the continuous process migrations that reduce considerably the data reuse. In this experiment, we have quantified these process migrations in 13 migrations per second on average.

Second, running with IRIX+SETC improves the speedup achieved (because of the effect commented in Section 3). However, the same scheduling problems of IRIX are not solved.

Third, notice that important improvements are obtained when running with JNE. This is caused by two factors: the inherent benefits of using JNE demonstrated in Section 6.1, and the action of the JNE scheduler in a multiprogrammed workload. In this case, the JNE scheduler binds kernel threads to processors, avoiding unnecessary process migrations and allowing more data reuse. In addition to this, the use of equitable policies like DSS makes possible that all applications instances in the workload get resources, avoiding application starvation or very unbalanced executions.

Considering the observed behavior, the only question is why application instances running with JNE in the workload do not achieve the speedup of their counterparts running alone. The answer to this question is the interference produced when running in *cpusets* as mentioned at the beginning of this section.

Finally, notice that in a non-overloaded system it is not important if applications are malleable, because there are enough resources to satisfy all the requests. Therefore, it is not necessary that applications adapt themselves.

These conclusions are consolidated in Table 2, which summarizes the observed performance degradation for each application instance in this workload with respect to best standalone execution. Performance degradation is calculated dividing the best application standalone execution time by the average execution time of an application instance in a workload.

Table 2. Performance degradation of each application instance in the first workload vs. the best standalone execution

Application	IRIX	IRIX + SETC	JNE not mall	JNE mall
LUFact 2 JTh	0,70	0,85	0,77	0,82
SOR 4 JTh	0,59	0,67	0,77	0,83
LUFact 4 JTh	0,62	0,71	0,93	0,95
SOR 6 JTh	0,53	0,64	0,81	0,74

For our third set of experiments, we have defined a workload composed of an instance of LUFact with concurrency level of 4 threads, an instance of SOR with concurrency level of 8 threads, an instance of LUFact with concurrency level of 8 threads and an instance of SOR with concurrency level of 12 threads. These applications instances are simultaneously started on a *cpuset* with 16 processors, and they are continuously restarted until one of them is repeated 10 times. Notice that, the maximum load is 32, which is higher than the number of processors available in the *cpuset*, so the system is overloaded. With this experiment we intend to evaluate the performance of JOMP malleable applications when running in an overloaded multiprogrammed environment.

Figure 9 draws the speedup for each application instance in the workload relative to the sequential version. All the series have the same meaning as in the previous workload.

Since the system is overloaded, each application instance will probably receive fewer processors than requested (as many processors as assigned by the DSS policy in the JNE scheduler). Therefore, the speedup should be the same as if executed alone in the *cpuset* with the number of processors allocated by the JNE scheduler.

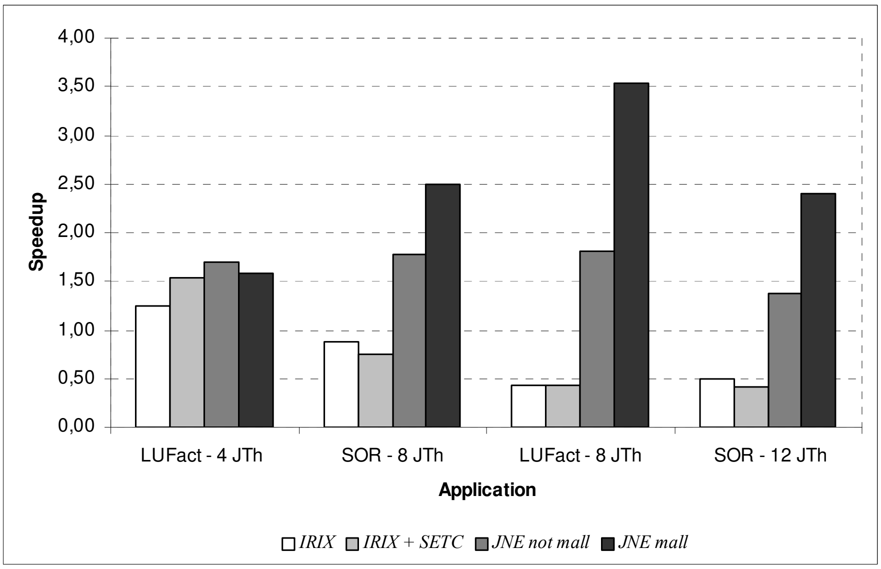


Figure 9. Speedup of each application instance in the second workload

All the conclusions exposed for the first workload are valid also in this case. In addition, some additional considerations must to be taken into account. First, the continuous process migrations when executing with IRIX have been incremented even more (43,9 process migrations per second on average).

In addition to this, notice that the IRIX scheduling causes a noticeable unbalanced execution (benefits some applications and damages others). For example, in this case LUFact with concurrency level of 4 threads is receiving proportionally more resources than the other application instances. For this reason, its performance degradation is lower.

When running with JNE, the action of the JNE scheduler in an overloaded multiprogrammed workload is even more important. A rational use of the resources allows the reduction of processor migrations (0,9 process migrations per second on average) allowing better locality exploitation and a balanced execution of all the application instances of the workload.

Like in the first workload, the interference produced when running in *cpusets* causes applications instances not to achieve the expected speedup. Besides, other factors as processor preemptions overhead or the processor distribution algorithm itself, can influence the speedup obtained.

Notice that, in an overloaded system it is very important if applications are malleable, because there are not enough resources to satisfy all the requests. Malleability reduces the number of Java threads created by the application thus reducing the overheads incurred in the parallel execution and management of threads. Figure 9 shows that the speedup achieved by applications approaches the speedup of using half the number of threads (as assigned by the DSS policy in this scenario).

Table 3. Performance degradation of each application instance in the second workload vs. the best standalone execution

Application	IRIX	IRIX + SETC	JNE not mall	JNE mall
LUFact 4 JTh	0,30	0,37	0,40	0,38
SOR 8 JTh	0,19	0,17	0,40	0,57
LUFact 8 JTh	0,08	0,08	0,34	0,66
SOR 12 JTh	0,08	0,07	0,25	0,43

Table 3 summarizes the observed performance degradation for each application instance in the second workload with respect to best standalone execution. Notice that the results confirm the benefit obtained when running multiprogrammed workloads with JNE, and the convenience of using of malleable applications able to adapt themselves to the available resources.

6.2.2 Non-malleable applications

For our fourth set of experiments, we have defined a workload composed of an instance of Series with concurrency level of 2 threads, an instance of Crypt with concurrency level of 4 threads, an instance of Series with concurrency level of 4 threads and an instance of Crypt with concurrency level of 6 threads. These applications instances are simultaneously started inside a *cpuset* with 16 processors, and they are continuously restarted until one of them is repeated 10 times. With this experiment we intend to evaluate the performance of JOMP non-malleable applications when running in a non-overloaded multiprogrammed environment.

Figure 10 draws the speedup obtained for each application instance in the workload relative to the sequential version. All the series have the same meaning as in the previous workload.

Notice that with non-malleable applications, the adaptation to the available resources is done only once, at the beginning of the only parallel region, and maintained during the entire region. This fact can lead to have unused processors if the application receives more processors while it is executing inside the parallel region, because at this point the application cannot generate new parallelism to run at these processors. In order to avoid this situation, we have decided that non-malleable applications use the JNE interface to adapt their concurrency level to the double of the available resources.

Since the system is not overloaded, each application instance should be able to run with as many processors as requested. Therefore, the speedup should be the same as if executed alone in the *cpuset*. The results obtained in this workload verify this theory.

Notice that, as we said in Section 6.1, in this kind of applications process migrations are not critical for performance (when running with IRIX+SETC we have measured 6,4 migrations per second on average without detecting any performance degradation).

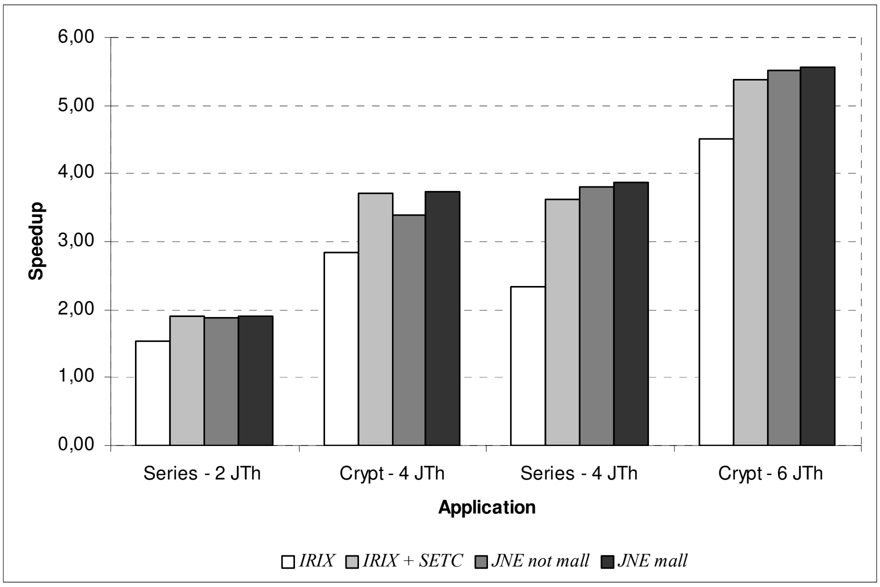


Figure 10. Speedup of each application instance in the third workload

This experiment confirms that in a non-overloaded system it is not important if applications adapt their behavior to the available resources, because there are enough resources to satisfy all the requests.

Table 4. Performance degradation of each application instance in the third workload vs. the best standalone execution

Application	IRIX	IRIX + SETC	JNE not mall	JNE mall
Series 2 JTh	0,78	0,97	0,93	0,93
Crypt 4 JTh	0,74	0,97	0,90	0,99
Series 4 JTh	0,62	0,95	0,98	0,99
Crypt 6 JTh	0,79	0,94	0,98	0,99

These conclusions are consolidated in Table 4, which summarizes the observed performance degradation for each application instance in the third workload with respect to best standalone execution.

For our fifth set of experiments, we have defined a workload composed of an instance of Series with concurrency level of 4 threads, an instance of Crypt with concurrency level of 8 threads, an instance of Series with concurrency level of 8 threads and an instance of Crypt with concurrency level of 12 threads. These applications instances are simultaneously started on a *cpuset* with 16 processors, and they are continuously restarted until one of them is repeated 10 times. Notice that, the system is overloaded. With this experiment we intend to evaluate the performance of JOMP non-malleable applications when running in an overloaded multiprogrammed environment.

Figure 11 draws the speedup for each application instance in the workload relative to the sequential version. All the series have the same meaning as in the previous workload.

Since the system is overloaded, each application instance will probably receive fewer processors than requested (as many processors as assigned by DSS policy in the JNE scheduler). Therefore, the speedup should be the same as if executed alone in the *cpuset* with the number of processors allocated by the JNE scheduler. The results obtained in this workload verify this theory and confirm that performance obtained when running with JNE and generating as parallelism as the double of the resources assigned (JNE mall) is comparable to the one obtained when running with the original system.

Finally, notice the performance degradation produced when running with JNE not mall, because the applications do not adapt to the available resources.

Table 5. Performance degradation of each application instance in the fourth workload vs. the best standalone execution

Application	IRIX	IRIX + SETC	JNE not mall	JNE mall
Series 4 JTh	0,60	0,63	0,49	0,49
Crypt 8 JTh	0,61	0,65	0,47	0,61
Series 8 JTh	0,49	0,47	0,50	0,49
Crypt 12 JTh	0,43	0,41	0,33	0,44

Table 5 shows the performance degradation of each application instance in the fourth workload with respect to best standalone execution.

7. CONCLUSIONS

In this paper we have shown how the interaction between the application and the underlying execution environment improves the performance of parallel Java applications on multiprogrammed shared memory multiprocessors.

The paper shows two scenarios. In the first one the application is able to inform the underlying execution environment about its concurrency level. As shown in our experimental results, the effect on performance of this communication is low when executing applications that create threads with a long lifetime.

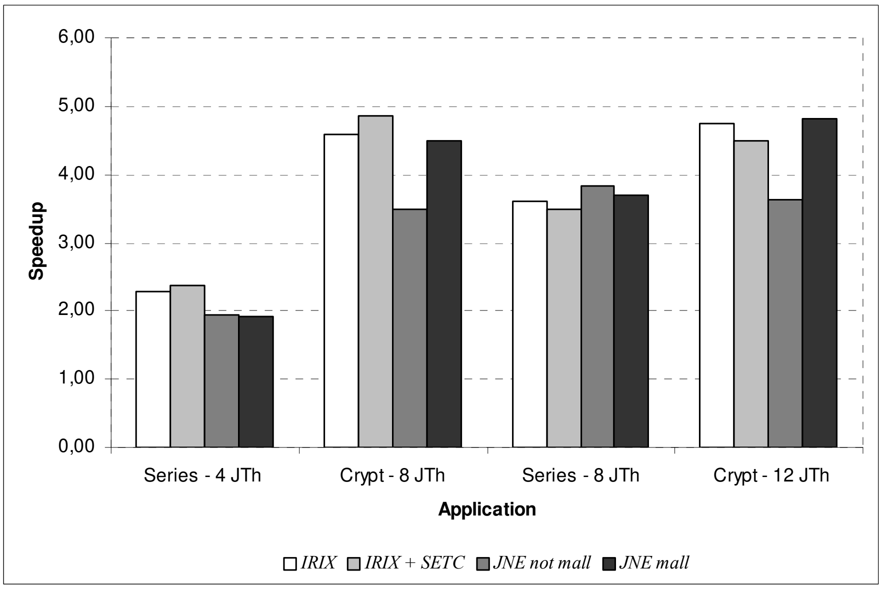


Figure 11. Speedup of each application instance in the fourth workload

In the second scenario, in addition to this communication path, the execution environment is also able to inform the application about the resource allocation decisions. The application is able to react to these decisions, changing the degree of parallelism that it is actually exploited from the application. Our experimental results show a noticeable impact on the final performance when applications are malleable.

Although this scenario is based on malleable applications, in the paper we have demonstrated that is also possible to maintain the efficiency of non-malleable applications.

Both mechanisms are implemented in the *Java Nanos Environment* (JNE), part of the *Barcelona Java Suite* (BJS) research environment, currently under development at CEPBA. JNE provides a solid investigation platform to improve the performance of parallel Java applications on parallel architectures.

8. ACKNOWLEDGMENTS

We acknowledge the European Center for Parallelism of Barcelona (CEPBA) for supplying the computing resources for our experiments. This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIC2001-0995-C02-01 and by Direcció General de Recerca of the Generalitat de Catalunya under grant 2001FI 00694 UPC APTIND. For additional information about the authors, please visit the Barcelona eDragon site at CEPBA (*http://www.cepba.upc.es/eDragon*).

9. REFERENCES

[1] B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, S.F. Hummel, D. Lieber, V. Litvinov, M.F. Mergen, T. Ngo, J.R. Russell, V. Sarkar, M.J. Serrano, J.C. Shepherd, S.E. Smith, V.C. Sreedhar, H. Srinivasan, and J. Whaley. "The Jalapeño Virtual Machine". IBM System Journal, Vol 39, No 1, February 2000.

[2] M. Bull, L.A. Smith, L. Pottage and R. Freeman. "Benchmarking Java against C and Fortran for Scientific Applications". ACM Java Grande/ISCOPE Conference, Stanford, California (USA), pp. 97-105. June 2001.

[3] M. Bull and M.E. Kambites. "JOMP -- an OpenMP-like Interface for Java". 2000 ACM Java Grande Conference, pp. 44-53, San Francisco (USA). June 2000.

[4] J. Corbalan, X. Martorell and J. Labarta. “Performance-Driven Processor Allocation”. 4th Operating System Design and Implementation (OSDI'00), pp. 59-73, San Diego, California (USA). October 2000.

[5] J. Corbalan and J. Labarta. “Improving Processor Allocation through Run-Time Measured Efficiency”. 15th International Parallel and Distributed Processing Symposium (IPDPS'2001), pp. 74-80, San Francisco, California (USA). April 2001.

[6] D. Feitelson. “Job Scheduling in Multiprogrammed Parallel Systems”. IBM Research Report 19790, August 1997.

[7] International Business Machines Corporation. “AIX V4.3.3 Workload Manager. Technical Reference”. February 2000 update, February 2000.

[8] J. Guitart, X. Martorell, J. Torres and E. Ayguadé. “Efficient Execution of Parallel Java Applications”. 3rd Annual Workshop on Java for High Performance Computing (part of the 15th ACM International Conference on Supercomputing ICS'01), pp. 31-35, Sorrento (Italy). June 17, 2001

[9] Java Grande Forum Benchmarks Suite. *http://www.epcc.ed.ac.uk/javagrande/*

[10] X. Martorell, J. Corbalan, D.S. Nikolopoulos, N. Navarro, E.D. Polychronopoulos, T.S. Papatheodorou and J. Labarta. "A Tool to Schedule Parallel Applications on Multiprocessors: the NANOS CPU Manager". 6th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'2000), in conjunction with the 14th IEEE International Parallel and Distributed Processing Symposium (IPDPS'2000), pp. 55-69, Cancun (Mexico). May 2000.

[11] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé. “A Library Implementation of the Nano Threads Programming Model”. 2nd EuroPar Conference, pp. 644-649, Lyon (France). August 1996.

[12] J. Oliver, E. Ayguadé, N. Navarro, J. Guitart, and J. Torres. “Strategies for Efficient Exploitation of Loop-level Parallelism in Java”. Concurrency and Computation: Practice and Experience (Java Grande 2000 Special Issue), Vol.13 (8-9), pp. 663-680. ISSN 1532-0634, July 2001.

[13] E.D. Polychronopoulos, X. Martorell, D. Nikolopoulos, J. Labarta, T.S. Papatheodorou and N. Navarro. “Kernel–level Scheduling for the Nano-Threads Programming Model”. 12th ACM International Conference on Supercomputing (ICS’98), pp. 337-344, Melbourne (Australia). July 1998.

[14] E.D. Polychronopoulos, D.S. Nikolopoulos, T.S. Papatheodorou, X. Martorell, J. Labarta and N. Navarro. “An Efficient Kernel-Level Scheduling Methodology for Multiprogrammed Shared Memory Multiprocessors”. 12th International Conference on Parallel and Distributed Computing Systems (PDCS’99), pp. 148-155, Fort Lauderdale, Florida (USA). August 18-20, 1999.

[15] “POSIX Threads”. IEEE POSIX 1003.1c Standard, 1995.

[16] Silicon Graphics Inc. “IRIX Admin: Resource Administration”. Document number 007-3700-005, *http://techpubs.sgi.com*. 2000.

[17] Silicon Graphics Inc. “Origin200 and Origin2000 Technical Report”, 1996.

[18] I. Subramanian, C. McCarthy, M. Murphy. “Meeting Performance Goals with the HP-UX Workload Manager”, 1st Workshop on Industrial Experiences with Systems Software (WIESS 2000), pp. 79-80, Usenix Association, San Diego, California (USA). October 2000.

[19] Sun Microsystems. “Java Native Interface”. February 1998. *http://java.sun.com/products/jdk/1.2/docs/guide/jni/*

[20] Sun Microsystems. “Solaris Resource Manager[tm] 1.0: Controlling System Resources Effectively”. 2000. *http://www.sun.com/software/white-papers/wp-srm/*